Smart pointers

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school

Łukasz Ziobroń & Bartosz Szurgot – autorzy prezentacji

Łukasz Ziobroń

Not only programming experience:

- C++ and Python developer @ Nokia & Credit Suisse
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Webmaster (HTML, PHP, CSS) @ StarCraft Area

Training experience:

- C++ trainings @ Coders School
- Practial Aspects Of Software Engineering @ PWr, UWr
- Nokia Academy @ Nokia
- Internal corporate trainings

Public speaking experience:

- Academic Championships in Team Programming
- code::dive conference
- code::dive community

Agenda

1. Smart pointers
2. Best practices
3. Implementation details
4. Efficiency

Smart pointers

Smart pointers

A smart pointer manages a pointer to a heap allocated object

- Deletes the pointed-to object at the right time
- `operator->()` call managed object methods
- `operator.()` call smart pointer methods
- smart pointer to a base class can hold a pointer to a derived class

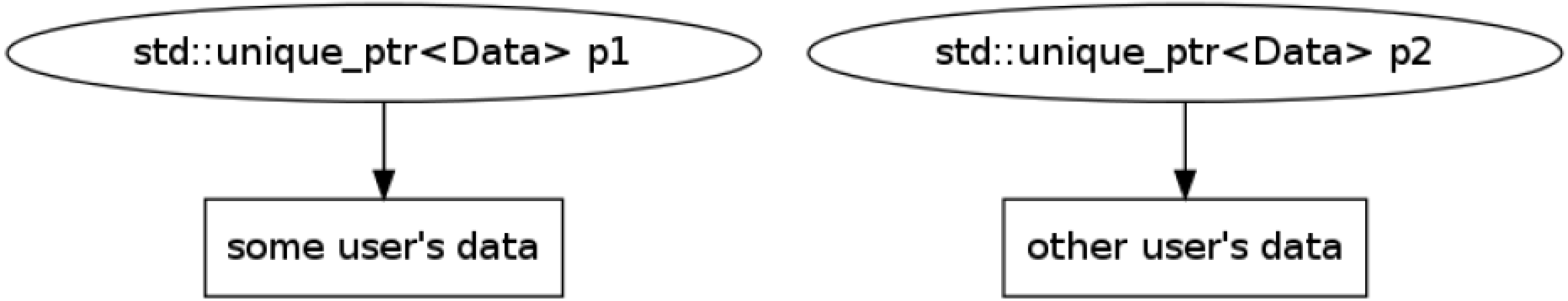
STL smart pointers:

- `std::unique_ptr<>`
- `std::shared_ptr<>`
- `std::weak_ptr<>`
- `std::auto_ptr<>` - removed in C++17

std::unique_ptr<>

Traits:

- one object == one owner
- destructor destroys the object
- copying not allowed
- moving allowed
- can use custom deleter



std::unique_ptr<> usage

- old style approach vs modern approach

```cpp
#include <iostream>  // old-style approach

struct Msg {
    int getValue() { return 42; }
};

Msg* createMsg() {
    return new Msg{};
}

int main()
{
    auto msg = createMsg();
    std::cout << msg->getValue();
    delete msg;
}
```

```cpp
#include <memory>  // modern approach
#include <iostream>

struct Msg {
    int getValue() { return 42; }
};

std::unique_ptr<Msg> createMsg() {
    return std::unique_ptr<Msg>(new Msg{});
}

int main()
{
    auto msg = createMsg();  // unique ownership
    std::cout << msg->getValue();
}
```

std::unique_ptr<> usage

- copying is not allowed, moving is allowed

```
std::unique_ptr<MyData> source(void);
void sink(std::unique_ptr<MyData> ptr);

void simpleUsage() {
    source();
    sink(source());
    auto ptr = source();
    // sink(ptr); ERROR
    sink(std::move(ptr));

    auto p1 = source();
    //auto p2 = p1; ERROR
    auto p2 = std::move(p1);
    //p1 = p2; ERROR
    p1 = std::move(p2);
}
```

```
void collections() {
    std::vector<std::unique_ptr<MyData>> v;
    v.push_back(source());

    auto tmp = source();
    //v.push_back(tmp); ERROR
    v.push_back(std::move(tmp));

    //sink(v[0]); ERROR
    sink(std::move(v[0]));
}
```

std::unique_ptr<> cooperation with raw pointers

- `get()` - returns a raw pointer without releasing the ownership
- `release()` - returns a raw pointer and release the ownership
- `reset()` - replaces the manager object
- `operator*()` - dereferences pointer to the managed object

```cpp
#include <memory>

void legacyInterface(int*) {}
void deleteResource(int* p) { delete p; }
void referenceInterface(int&) {}
```

```cpp
int main() {
    auto ptr = std::make_unique<int>(5);
    legacyInterface(ptr.get());
    deleteResource(ptr.release());
    ptr.reset(new int{10});
    referenceInterface(*ptr);
    ptr.reset(); // ptr is a nullptr
    return 0;
}
```

std::make_unique()

`std::make_unique()` is a factory function that produce unique_ptrs

- added in C++14 for symmetrical operations on unique and shared pointers
- avoids bare new expression

```cpp
#include <memory>

struct Msg {
    Msg(int i) : value(i) {}
    int value;
};

int main() {
    auto ptr1 = std::unique_ptr<Msg>(new Msg{5});
    auto ptr2 = std::make_unique<Msg>(5);
    return 0;
}
```

std::unique_ptr<T[]>

- During destruction
  - std::unique_ptr<T> calls delete
  - std::unique_ptr<T[]> calls delete[]
- std::unique_ptr<T[]> has additional operator[] for accessing array element
- Usually std::vector<T> is a better choice

```cpp
struct MyData {};
void processPointer(MyData* md) {}
void processElement(MyData md) {}
using Array = std::unique_ptr<MyData[]>;
```

```cpp
void use(void)
{
    Array tab{new MyData[42]};
    processPointer(tab.get());
    processElement(tab[13]);
}
```

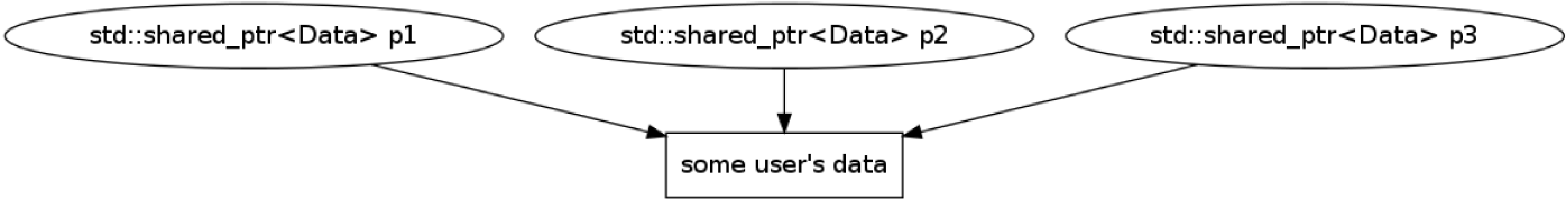
Exercise: ResourceD

1. Compile and run ResourceD application and check memory leaks under valgrind
2. Fix memory leaks with a proper usage of delete operator
3. Refactor the solution to use `std::unique_ptr<>`
4. Use `std::make_unique()`

std::shared_ptr<>

Traits:

- one object == multiple owners
- last referrer destroys the object
- copying allowed
- moving allowed
- can use custom deleter
- can use custom allocator



std::shared_ptr<> usage

- copying and moving is allowed

```
std::shared_ptr<MyData> source();
void sink(std::shared_ptr<MyData> ptr);

void simpleUsage() {
    source();
    sink(source());
    auto ptr = source();
    sink(ptr);
    sink(std::move(ptr));

    auto p1 = source();
    auto p2 = p1;
    p2 = std::move(p1);
    p1 = p2;
    p1 = std::move(p2);
}
```

```
void collections() {
    std::vector<std::shared_ptr<MyData>> v;
    v.push_back(source());

    auto tmp = source();
    v.push_back(tmp);
    v.push_back(std::move(tmp));

    sink(v[0]);
    sink(std::move(v[0]));
}
```

std::shared_ptr<> usage

```
#include <memory>
#include <map>
#include <string>

class Gadget {};
std::map<std::string, std::shared_ptr<Gadget>> gadgets;  // it wouldn't compile with C++03. Why?

void foo() {
    std::shared_ptr<Gadget> p1{new Gadget()};    // reference counter = 1
    {
        auto p2 = p1;                            // shared_ptr copy (reference counter == 2)
        gadgets.insert(make_pair("mp3", p2));    // shared_ptr copy (reference counter == 3)
        p2->use();
    }                                            // destruction of p2, reference counter = 2
}                                                // destruction of p1, reference counter = 1

gadgets.clear();                                 // reference counter = 0 - gadget is removed
```

std::shared_ptr<> cyclic dependencies

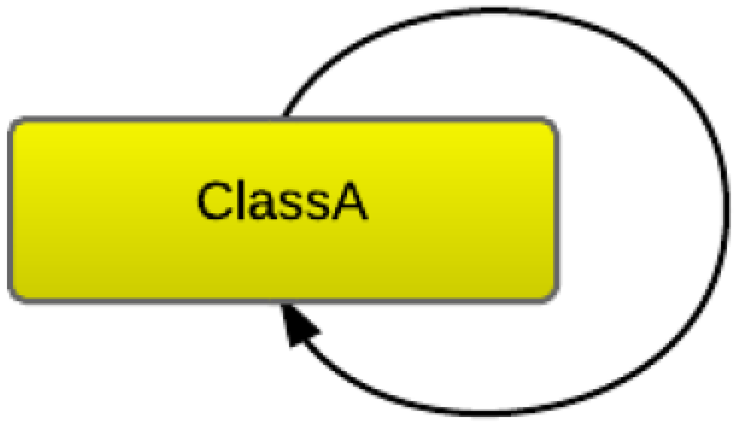
- What happens here?

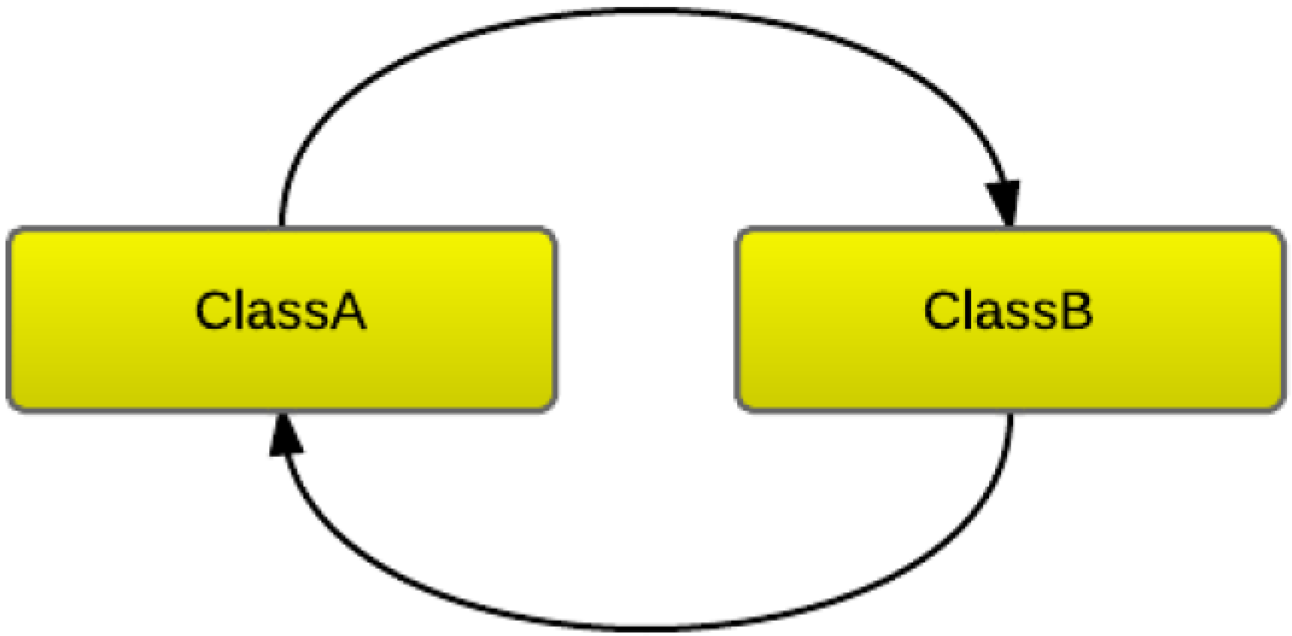
```
#include <memory>

struct Node {
    std::shared_ptr<Node> child;
    std::shared_ptr<Node> parent;
};

int main ( )
{
    auto root = std::shared_ptr<Node>(new Node);
    auto child = std::shared_ptr<Node>(new Node);
    root->child = child;
    child->parent = root;
 }
```

Memory leak

Cyclic dependencies

- Cyclic dependency is where you have class A with self-referencing member.
- Cyclic dependency is where you have two classes A and B where A has a reference to B which has a reference to A.
- How to fix it?

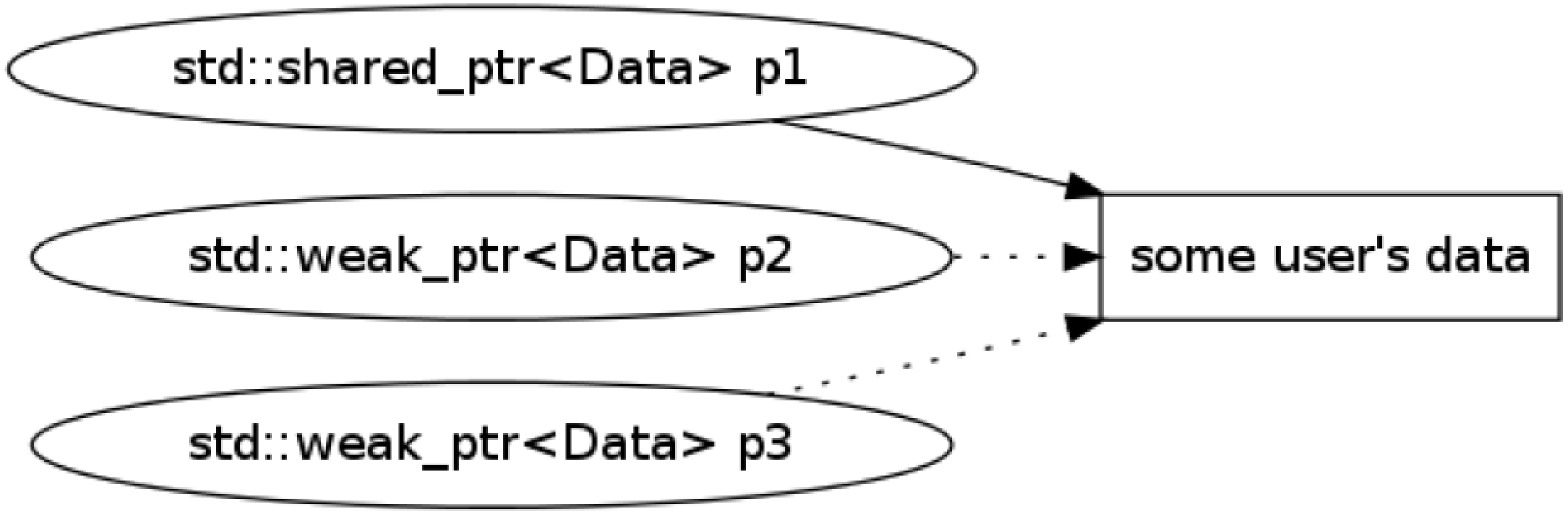




std::weak_ptr<> to the rescue

Traits:

- does not own an object
- observes only
- must be converted to `std::shared_ptr<>` to access the object
- can be created only from a `std::shared_ptr<>`



std::weak_ptr<> usage

```cpp
#include <memory>
#include <iostream>

struct Msg { int value; };

void checkMe(const std::weak_ptr<Msg> & wp) {
    auto p = wp.lock();
    if (p)
        std::cout << p->value << '\n';
    else
        std::cout << "Expired\n";
}

int main() {
    auto sp = std::shared_ptr<Msg>{new Msg{10}};
    auto wp = std::weak_ptr<Msg>{sp};
    checkMe(wp);
    sp.reset();
    checkMe(wp);
}
```

```
> ./a.out
10
Expired
```

std::shared_ptr<> cyclic dependencies

- How to solve this problem?

```cpp
#include <memory>

struct Node {
    std::shared_ptr<Node> child;
    std::shared_ptr<Node> parent;
};

int main ( )
{
    auto root = std::shared_ptr<Node>(new Node);
    auto child = std::shared_ptr<Node>(new Node);
    root->child = child;
    child->parent = root;
 }
```

Breaking cycle - solution

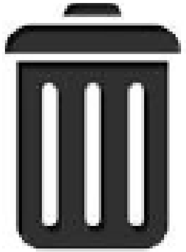
- Use std::weak_ptr<Node> in one direction

```cpp
#include <memory>

struct Node {
    std::shared_ptr<Node> child;
    std::weak_ptr<Node> parent;
};

int main ( )
{
    auto root = std::shared_ptr<Node>(new Node);
    auto child = std::shared_ptr<Node>(new Node);
    root->child = child;
    child->parent = root;
 }
```

```
==148== All heap blocks were freed -- no leaks are possible
```

std::auto_ptr<> - something to forget

- C++98 provided `std::auto_ptr<>`
- Few fixes in C++03
- Yet still it's easy to use incorrectly…
- Deprecated since C++11
- Removed since C++17
- Do not use it, use `std::unique_ptr<>` instead



Smart pointers - summary

- `#include <memory>`
- `std::unique_ptr<>` for exclusive ownership
- `std::shared_ptr<>` for shared ownership
- `std::weak_ptr<>` for observation and breaking cycles

Exercise: ResourceFactory

1. Compile and run ResourceFactory application
2. Put comments in places where you can spot some problems
3. How to remove elements from the collection `(vector<Resource*> resources)`?
4. Check memory leaks
5. Fix problems

Best practices

Best practices

- Rule of 0, Rule of 5
- Avoid explicit new
- Use `std::make_shared()` / `std::make_unique()`
- Copying `std::shared_ptr<>`
- Use references instead of pointers

Rule of 0, Rule of 5

- Rule of 5

  If you need to implement one of those functions:

  - destructor
  - copy constructor
  - copy assignment operator
  - move constructor
  - move assignment operator

  It probably means that you should implement them all, because you have manual resources management.

- Rule of 0

  If you use RAII wrappers on resources, you don't need to implement any of Rule of 5 functions.

Avoid explicit new

- Smart pointers eliminate the need to use `delete` explicitly
- To be symmetrical, do not use `new` as well
- Allocate using:
  - `std::make_unique()`
  - `std::make_shared()`

Use std::make_shared() / std::make_unique()

- What is a problem here?

```
struct MyData { int value; };
using Ptr = std::shared_ptr<MyData>;
void sink(Ptr oldData, Ptr newData);

void use(void) {
    sink(Ptr{new MyData{41}}, Ptr{new MyData{42}});
}
```

- Hint: this version is not problematic

```
struct MyData{ int value; };
using Ptr = std::shared_ptr<MyData>;
void sink(Ptr oldData, Ptr newData);

void use(void) {
    Ptr oldData{new MyData{41}};
    Ptr newData{new MyData{42}};
    sink(std::move(oldData), std::move(newData));
}
```

Use std::make_shared() / std::make_unique()

- `auto p = new MyData(10);` means
  - allocate sizeof(MyData) bytes
  - run MyData constructor
  - assign address of allocated memory to p

Order of evaluation of the operands of almost all C++ operators (including the order of evaluation of function arguments in a function-call expression and the order of evaluation of the subexpressions within any expression) **is unspecified**.

- How about two such operations?
  - (1) allocate sizeof(MyData) bytes
  - (2) run MyData constructor
  - (3) assign address of allocated memory to p
  - (1) allocate sizeof(MyData) bytes
  - (2) run MyData constructor
  - (3) assign address of allocated memory to p
- Unspecified order of evaluation means that order can be for example 1,2,1,2,3,3.
- What if second 2 throws an exception?

Use std::make_shared() / std::make_unique()

- `std::make_shared() / std::make_unique()` resolves this problem

```
struct MyData{ int value; };
using Ptr = std::shared_ptr<MyData>;
void sink(Ptr oldData, Ptr newData);

void use() {
    sink(std::make_shared<MyData>(41), std::make_shared<MyData>(42));
}
```

- Fixes previous bug
- Does not repeat a constructed type
- Does not use explicit new
- Optimizes memory usage (only for `std::make_shared()`)

Copying std::shared_ptr<>

```cpp
void foo(std::shared_ptr<MyData> p);

void bar(std::shared_ptr<MyData> p) {
    foo(p);
}
```

- requires counters incrementing / decrementing
- atomics / locks are not free
- will call destructors

Can be better?

Copying std::shared_ptr<>

```
void foo(const std::shared_ptr<MyData> & p);

void bar(const std::shared_ptr<MyData> & p) {
    foo(p);
}
```

- as fast as pointer passing
- no extra operations
- not safe in multithreaded applications

Use references instead of pointers

- What is the difference between a pointer and a reference?
  - reference cannot be empty
  - once assigned cannot point to anything else

- Priorities of usage (if possible):
  - `(const) T&`
  - `std::unique_ptr<T>`
  - `std::shared_ptr<T>`
  - `T*`

Exercise: List

Take a look at List.cpp file, where simple (and buggy) single-linked list is implemented.
`void add(Node* node)` method adds a new Node at the end of the list
`Node* get(const int value)` method iterate over the list and returns the first Node with matching value or `nullptr`

1. Compile and run List application
2. Fix memory leaks without introducing smart pointers
3. Fix memory leaks with smart pointers. What kind of pointers needs to be applied and why?
4. What happens when the same Node is added twice? Fix this problem.
5. (Optional) Create `EmptyListError` exception (deriving from `std::runtime_error`). Add throwing and catching it in a proper places.

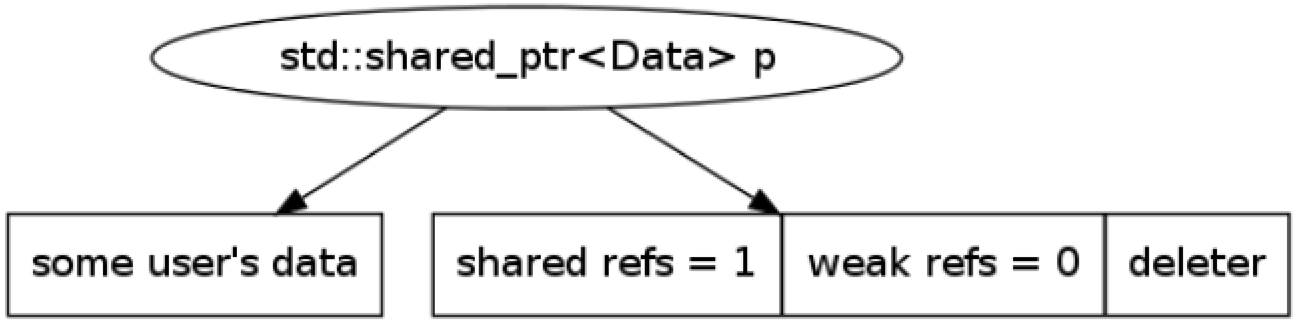
Implementation details

Implementation details – std::unique_ptr<>

- Just a holding wrapper
- Holds an object pointer
- Constructor copies a pointer
- Call proper delete in destructor
- No copying
- Moving means:
  - Copying original pointer to a new object
  - Setting source pointer to nullptr
- All methods are inline

Implementation details – std::shared_ptr<>

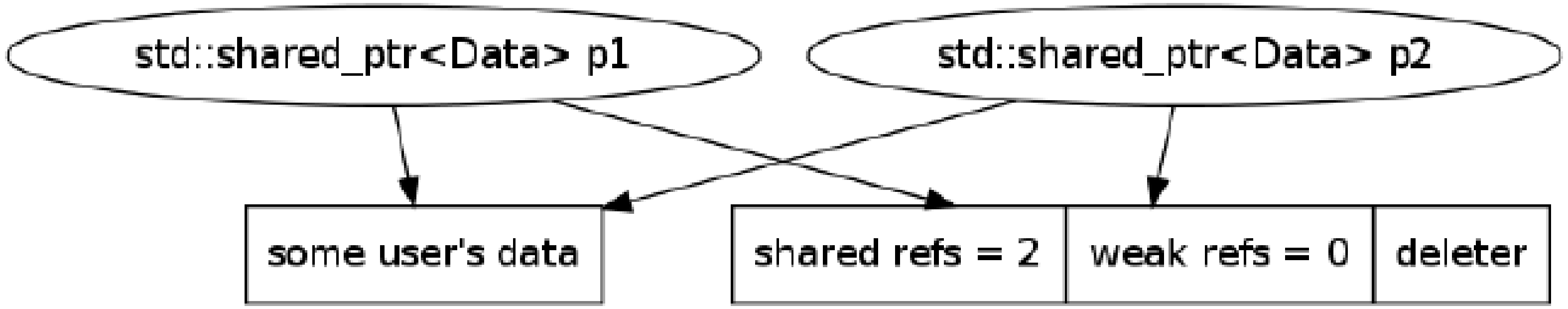
- Holds an object pointer
- Holds 2 reference counters:
  - shared pointers count
  - weak pointers count



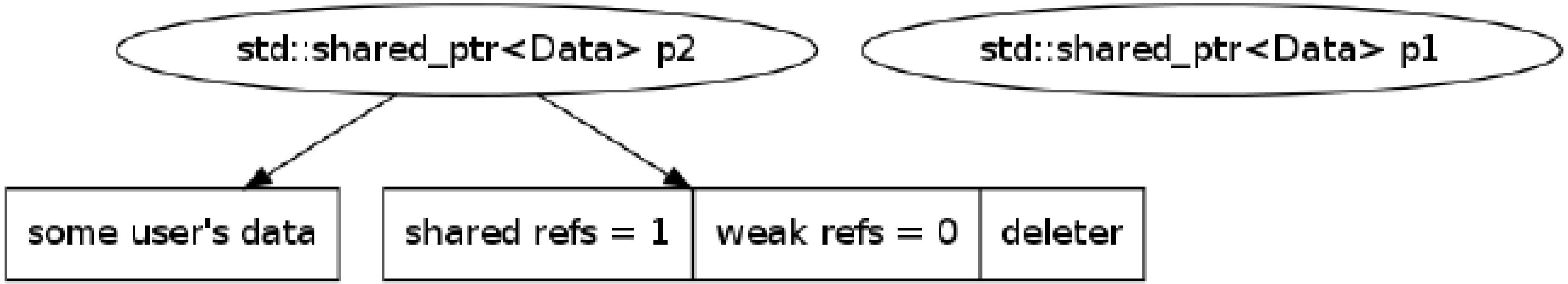
- Destructor:
  - decrements shared-refs
  - deletes user data when shared-refs == 0
  - deletes reference counters when shared-refs == 0 and weak-refs == 0
- Extra space for a deleter
- All methods are inline

Implementation details – std::shared_ptr<>

- Copying means:
  - Copying pointers to the target
  - Incrementing shared-refs

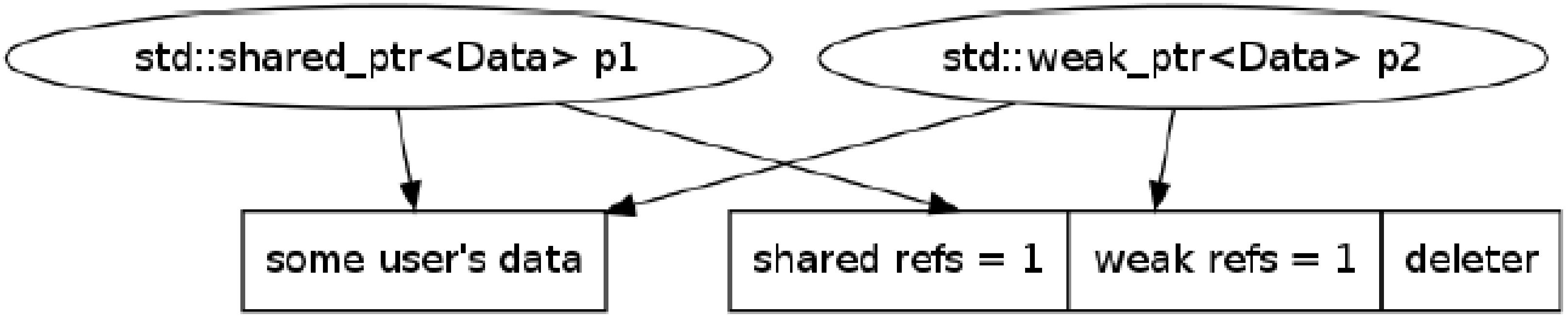


- Moving means:
  - Copying pointers to the target
  - Setting source pointers to nullptr



Implementation details – std::weak_ptr<>

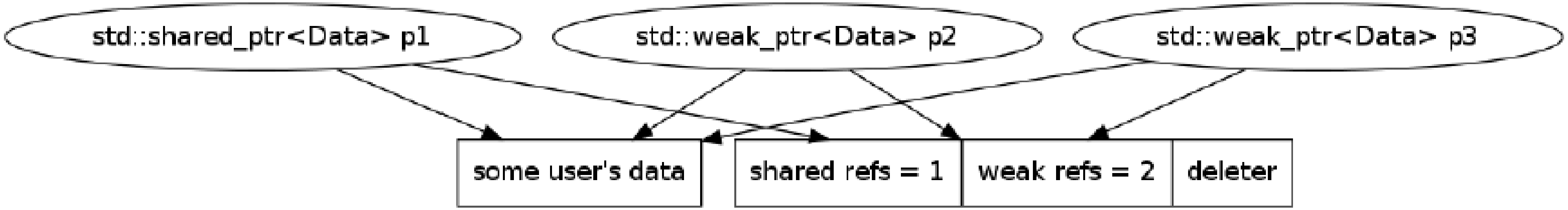
- Holds an object pointer
- Holds 2 reference counters:
  - shared pointers count
  - weak pointers count



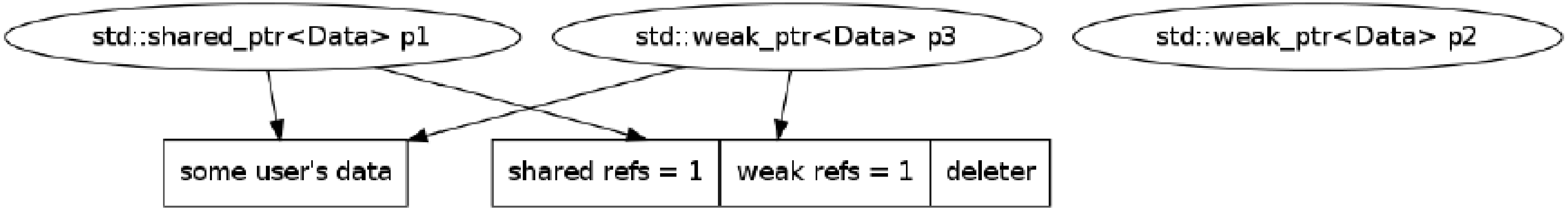
- Destructor:
  - decrements weak-refs
  - deletes reference counters when shared-refs == 0 and weak-refs == 0

Implementation details – std::weak_ptr<>

- Copying means:
  - Copying pointers to the target
  - Incrementing weak-refs

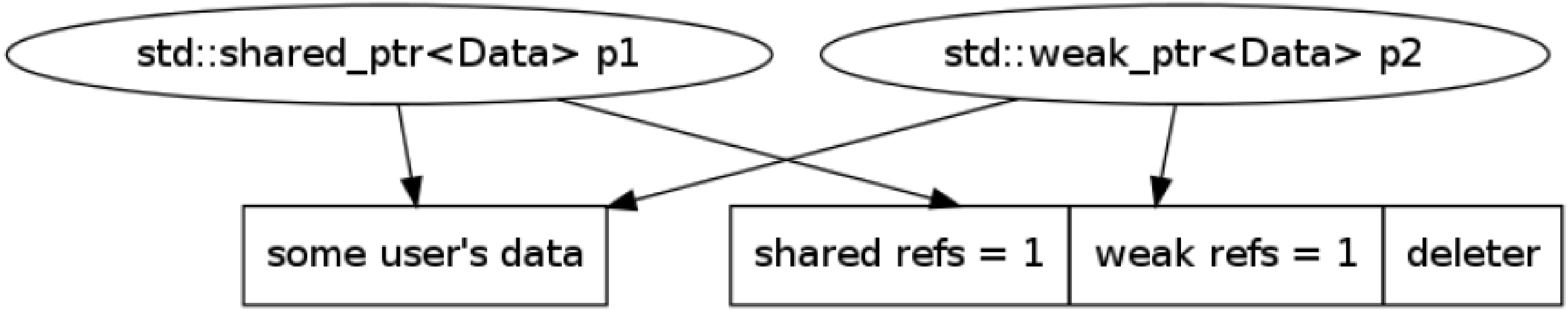


- Moving means:
  - Copying pointers to the target
  - Setting source pointers to nullptr

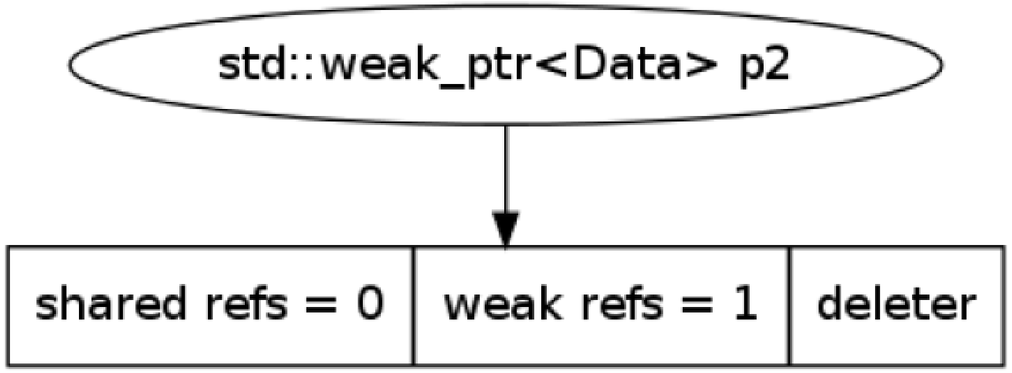


std::weak_ptr<> + std::shared_ptr<>

- Having a shared pointer and a weak pointer

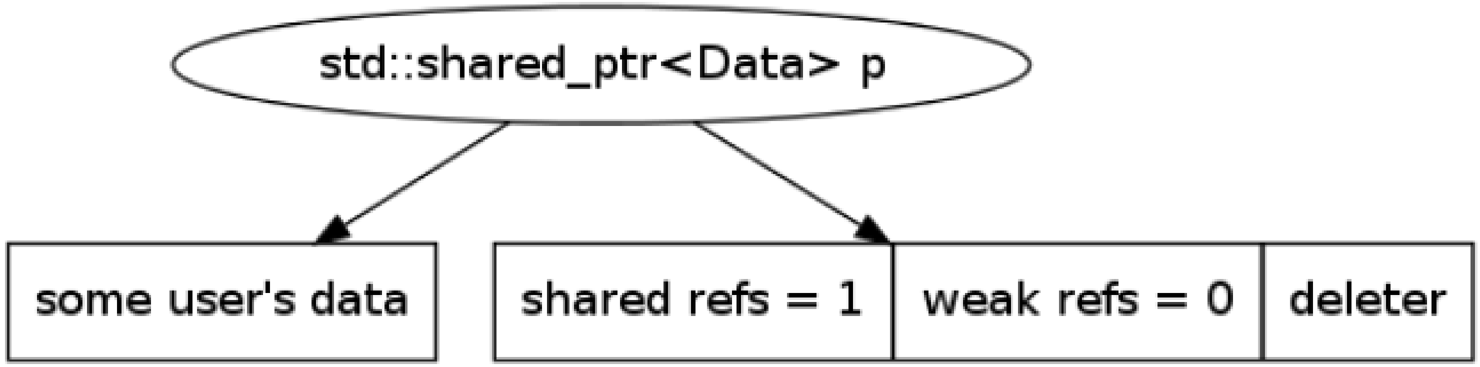


- After removing the shared pointer

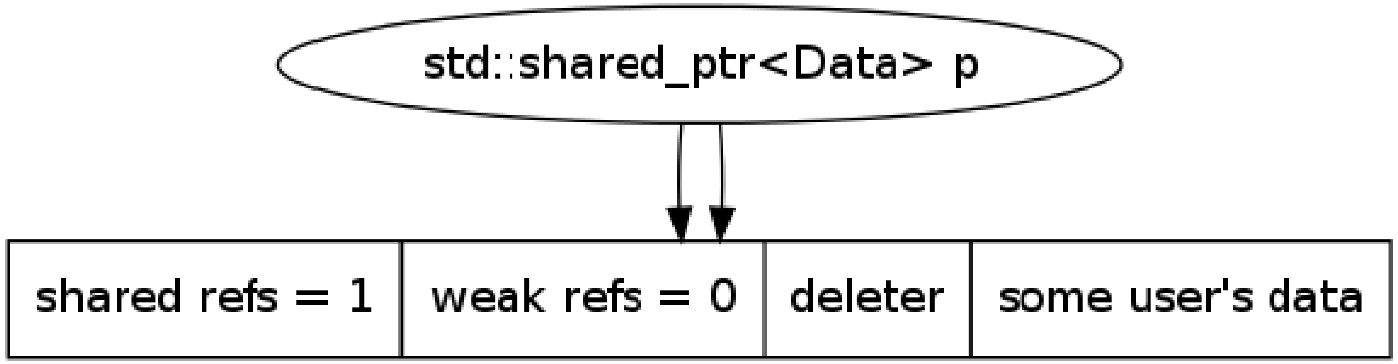


Making a std::shared_ptr<>

- `std::shared_ptr<Data> p{new Data};`



- `auto p = std::make_shared<Data>();`
  - Less memory (most likely)
  - Only one allocation
  - Cache-friendly



Efficiency

Raw pointer

```cpp
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<Data *> v;
    v.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        auto p = new Data;
        v.push_back(std::move(p));
    }
    for (auto p: v)
        delete p;
}
```

Unique pointer

```cpp
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<std::unique_ptr<Data>> v;
    v.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        std::unique_ptr<Data> p{new Data};
        v.push_back(std::move(p));
    }
}
```

Shared pointer

```cpp
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<std::shared_ptr<Data>> v;
    v.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        std::shared_ptr<Data> p{new Data};
        v.push_back(std::move(p));
    }
}
```

Shared pointer – make_shared

```cpp
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<std::shared_ptr<Data>> v;
    v.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        auto p = std::make_shared<Data>();
        v.push_back(std::move(p));
    }
}
```

Weak pointer

```cpp
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<std::shared_ptr<Data>> vs;
    std::vector<std::weak_ptr<Data>> vw;
    vs.reserve(size);
    vw.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        std::shared_ptr<Data> p{new Data};
        std::weak_ptr<Data> w{p};
        vs.push_back(std::move(p));
        vw.push_back(std::move(w));
    }
}
```

Measurements

- gcc-4.8.2
- compilation with –std=c++11 –O3 –DNDEBUG
- measuring with:
  - time (real)
  - htop (mem)
  - valgrind (allocations count)

test name	time [s]	allocations	memory [MB]
raw pointer	0.54	10 000 001	686
unique pointer	0.56	10 000 001	686
shared pointer	1.00	20 000 001	1072
make shared	0.76	10 000 001	914
weak pointer	1.28	20 000 002	1222

Conclusions

- RAII
  - acquire resource in constructor
  - release resource in destructor
- Rule of 5, Rule of 0
- Smart pointers:
  - `std::unique_ptr` – primary choice, no overhead, can convert to std::shared_ptr
  - `std::shared_ptr` – introduces memory and runtime overhead
  - `std::weak_ptr` – breaking cycles, can convert to/from std::shared_ptr
- Create smart pointers with `std::make_shared()` and `std::make_unique()`
- Raw pointer should mean „access only” (no ownership)
- Use reference instead of pointers if possible

Post-work

1. Transform the list from List.cpp into double-linked list. You should implement:
   - inserting Nodes at the beginning of the list
   - searching elements from the backward

   Apply proper smart pointers for the reverse direction.

2. Implement your own unique_ptr. Requirements:
   - Templatized (should hold a pointer to a template type)
   - RAII (acquire in constructor, release in destructor)
   - Copying not allowed
   - Moving allowed
   - Member functions: `operator*()`, `operator->()`, `get()`, `release()`, `reset()`

3. Read one of these articles on move semantics:
   - Semantyka przenoszenia (in Polish)
   - Move semantics and rvalue references in C++11 (in English)

CODERS SCHOOL

https://coders.school

Łukasz Ziobroń
lukasz@coders.school